Protocolos de Meta-Objecto

António Menezes Leitão

8 de Abril de 2008

1. CLOS
   - História
   - Funções Genéricas
   - Classes

CLOS-Common Lisp Object System

Raizes

- 1980: Flavors - TI Explorer
- 1985: NewFlavors - Symbolics
- 1986: Loops (Lisp Object Oriented Programming System), CommonLoops - Xerox Lisp Machines
- 1986: ObjectLisp - LMI Lambda
- 1987: Common Objects - HP

Características

- Funções Genéricas, Despacho Múltiplo.
- Classes, Herança Múltipla.
- Meta-Objectos, Protocolos.

Invocação de Funções

Programação Funcional e Imperativa

```
(foo a b)
⇓
(call (function 'foo) a b)
```

Programação Orientada a Objectos - Despacho Simples

```
(foo a b) ⇔ a.foo(b)
⇓
(call (function 'foo (type-of a)) a b)
```

Programação Orientada a Objectos - Despacho Múltiplo

```
(foo a b)
⇓
(call (function 'foo (type-of a) (type-of b)) a b)
```

Despacho Múltiplo

Adicionar Entidades

```
(defgeneric add (x y))
```

Despacho Múltiplo

Adicionar Entidades

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))
```

Despacho Múltiplo

Adicionar Entidades

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))

;;Testing
> (add 1 3)
```

Despacho Múltiplo

Adicionar Entidades

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))

;;Testing
> (add 1 3)
4
```

Despacho Múltiplo

Adicionar Entidades

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))

;;Testing
> (add 1 3)
4
> (add (vector 1 2) (vector 3 4))
```

Despacho Múltiplo

Adicionar Entidades

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))

;;Testing
> (add 1 3)
4
> (add (vector 1 2) (vector 3 4))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2) #(3 4)) of classes
(VECTOR VECTOR)
```

Despacho Múltiplo

Adicionar Entidades

```
(defgeneric add (x y))

(defmethod add ((x number) (y number))
  (+ x y))

;;Testing
> (add 1 3)
4
> (add (vector 1 2) (vector 3 4))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2) #(3 4)) of classes
(VECTOR VECTOR)

(defmethod add ((x array) (y array))
  (assert (equal (array-dimensions x) (array-dimensions y)))
  (let ((z (make-array (array-dimensions x))))
    (dotimes (i (array-total-size x))
      (setf (row-major-aref z i)
            (add (row-major-aref x i) (row-major-aref y i))))
    z))
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2 3) (vector 4 5 6))
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2 3) (vector 4 5 6))
#(5 7 9)
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2 3) (vector 4 5 6))
#(5 7 9)

> (add (make-array '(2 3) :initial-contents '((1 2 3) (4 5 6)))
       (make-array '(2 3) :initial-element 10))
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2 3) (vector 4 5 6))
#(5 7 9)

> (add (make-array '(2 3) :initial-contents '((1 2 3) (4 5 6)))
       (make-array '(2 3) :initial-element 10))
#2A((11 12 13) (14 15 16))
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2 3) (vector 4 5 6))
#(5 7 9)

> (add (make-array '(2 3) :initial-contents '((1 2 3) (4 5 6)))
       (make-array '(2 3) :initial-element 10))
#2A((11 12 13) (14 15 16))

> (add (vector (vector 1 2) 3)
       (vector (vector 3 4) 5))
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2 3) (vector 4 5 6))
#(5 7 9)

> (add (make-array '(2 3) :initial-contents '((1 2 3) (4 5 6)))
       (make-array '(2 3) :initial-element 10))
#2A((11 12 13) (14 15 16))

> (add (vector (vector 1 2) 3)
       (vector (vector 3 4) 5))
#(#(4 6) 8)
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2 3) (vector 4 5 6))
#(5 7 9)

> (add (make-array '(2 3) :initial-contents '((1 2 3) (4 5 6)))
       (make-array '(2 3) :initial-element 10))
#2A((11 12 13) (14 15 16))

> (add (vector (vector 1 2) 3)
       (vector (vector 3 4) 5))
#(#(4 6) 8)

> (add (vector 1 2) 3)
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2 3) (vector 4 5 6))
#(5 7 9)

> (add (make-array '(2 3) :initial-contents '((1 2 3) (4 5 6)))
       (make-array '(2 3) :initial-element 10))
#2A((11 12 13) (14 15 16))

> (add (vector (vector 1 2) 3)
       (vector (vector 3 4) 5))
#(#(4 6) 8)

> (add (vector 1 2) 3)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2) 3) of classes
(VECTOR FIXNUM)
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2 3) (vector 4 5 6))
#(5 7 9)

> (add (make-array '(2 3) :initial-contents '((1 2 3) (4 5 6)))
       (make-array '(2 3) :initial-element 10))
#2A((11 12 13) (14 15 16))

> (add (vector (vector 1 2) 3)
       (vector (vector 3 4) 5))
#(#(4 6) 8)

> (add (vector 1 2) 3)
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#(1 2) 3) of classes
(VECTOR FIXNUM)
```

```
(defmethod add ((x array) (y t))
  (add x
       (make-array (array-dimensions x) :initial-element y)))
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2) 3)
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2) 3)
#(4 5)
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2) 3)
#(4 5)

> (add 1 (make-array '(2 2) :initial-contents '((1 2) (3 4))))
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2) 3)
#(4 5)

> (add 1 (make-array '(2 2) :initial-contents '((1 2) (3 4))))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (1 #2A((1 2) (3 4))) of
classes (FIXNUM ARRAY)
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2) 3)
#(4 5)

> (add 1 (make-array '(2 2) :initial-contents '((1 2) (3 4))))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (1 #2A((1 2) (3 4))) of
classes (FIXNUM ARRAY)

(defmethod add ((x t) (y array))
  (add (make-array (array-dimensions y) :initial-element x)
       y))
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2) 3)
#(4 5)

> (add 1 (make-array '(2 2) :initial-contents '((1 2) (3 4))))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (1 #2A((1 2) (3 4))) of
classes (FIXNUM ARRAY)

(defmethod add ((x t) (y array))
  (add (make-array (array-dimensions y) :initial-element x)
       y))

> (add 1 (make-array '(2 2) :initial-contents '((1 2) (3 4))))
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2) 3)
#(4 5)

> (add 1 (make-array '(2 2) :initial-contents '((1 2) (3 4))))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (1 #2A((1 2) (3 4))) of
classes (FIXNUM ARRAY)

(defmethod add ((x t) (y array))
  (add (make-array (array-dimensions y) :initial-element x)
       y))

> (add 1 (make-array '(2 2) :initial-contents '((1 2) (3 4))))
#2A((2 3) (4 5))
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2) 3)
#(4 5)

> (add 1 (make-array '(2 2) :initial-contents '((1 2) (3 4))))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (1 #2A((1 2) (3 4))) of
classes (FIXNUM ARRAY)

(defmethod add ((x t) (y array))
  (add (make-array (array-dimensions y) :initial-element x)
       y))

> (add 1 (make-array '(2 2) :initial-contents '((1 2) (3 4))))
#2A((2 3) (4 5))

> (add "12" "34")
```

Despacho Múltiplo

Adicionar Entidades

```
> (add (vector 1 2) 3)
#(4 5)

> (add 1 (make-array '(2 2) :initial-contents '((1 2) (3 4))))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (1 #2A((1 2) (3 4))) of
classes (FIXNUM ARRAY)

(defmethod add ((x t) (y array))
  (add (make-array (array-dimensions y) :initial-element x)
       y))

> (add 1 (make-array '(2 2) :initial-contents '((1 2) (3 4))))
#2A((2 3) (4 5))

> (add "12" "34")
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (#\1 #\3) of classes
(CHARACTER CHARACTER)
```

Despacho Múltiplo

Adicionar Entidades

```
(defmethod add ((x string) (y t))
  (add (read-from-string x) y))

(defmethod add ((x t) (y string))
  (add x (read-from-string y)))
```

Despacho Múltiplo

Adicionar Entidades

```
(defmethod add ((x string) (y t))
  (add (read-from-string x) y))

(defmethod add ((x t) (y string))
  (add x (read-from-string y)))

> (add "12" "34")
```

Despacho Múltiplo

Adicionar Entidades

```
(defmethod add ((x string) (y t))
  (add (read-from-string x) y))

(defmethod add ((x t) (y string))
  (add x (read-from-string y)))

> (add "12" "34")
46
```

Despacho Múltiplo

Adicionar Entidades

```
(defmethod add ((x string) (y t))
  (add (read-from-string x) y))

(defmethod add ((x t) (y string))
  (add x (read-from-string y)))

> (add "12" "34")
46
> (add (vector "123" "4") 5)
```

Despacho Múltiplo

Adicionar Entidades

```
(defmethod add ((x string) (y t))
  (add (read-from-string x) y))

(defmethod add ((x t) (y string))
  (add x (read-from-string y)))

> (add "12" "34")
46
> (add (vector "123" "4") 5)
#(128 9)
```

Despacho Múltiplo

Adicionar Entidades

```
(defmethod add ((x string) (y t))
  (add (read-from-string x) y))

(defmethod add ((x t) (y string))
  (add x (read-from-string y)))

> (add "12" "34")
46
> (add (vector "123" "4") 5)
#(128 9)
> (add (vector 1 2 3) (list 4 5 6))
```

Despacho Múltiplo

Adicionar Entidades

```
(defmethod add ((x string) (y t))
  (add (read-from-string x) y))

(defmethod add ((x t) (y string))
  (add x (read-from-string y)))

> (add "12" "34")
46
> (add (vector "123" "4") 5)
#(128 9)
> (add (vector 1 2 3) (list 4 5 6))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (1 (4 5 6)) of classes
(FIXNUM CONS)
```

Despacho Múltiplo

Adicionar Entidades

```
(defmethod add ((x string) (y t))
  (add (read-from-string x) y))

(defmethod add ((x t) (y string))
  (add x (read-from-string y)))

> (add "12" "34")
46
> (add (vector "123" "4") 5)
#(128 9)
> (add (vector 1 2 3) (list 4 5 6))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (1 (4 5 6)) of classes
(FIXNUM CONS)

(defmethod add ((x vector) (y list))
  (add x (coerce y 'vector)))
```

Despacho Múltiplo

Adicionar Entidades

```
(defmethod add ((x string) (y t))
  (add (read-from-string x) y))

(defmethod add ((x t) (y string))
  (add x (read-from-string y)))

> (add "12" "34")
46
> (add (vector "123" "4") 5)
#(128 9)
> (add (vector 1 2 3) (list 4 5 6))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (1 (4 5 6)) of classes
(FIXNUM CONS)

(defmethod add ((x vector) (y list))
  (add x (coerce y 'vector)))

> (add (vector 1 2 3) (list 4 5 6))
```

Despacho Múltiplo

Adicionar Entidades

```
(defmethod add ((x string) (y t))
  (add (read-from-string x) y))

(defmethod add ((x t) (y string))
  (add x (read-from-string y)))

> (add "12" "34")
46
> (add (vector "123" "4") 5)
#(128 9)
> (add (vector 1 2 3) (list 4 5 6))
No methods applicable for generic function
#<STANDARD-GENERIC-FUNCTION ADD> with args (1 (4 5 6)) of classes
(FIXNUM CONS)

(defmethod add ((x vector) (y list))
  (add x (coerce y 'vector)))

> (add (vector 1 2 3) (list 4 5 6))
#(5 7 9)
```

Especialização em Instâncias

Função factorial

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n > 0 \end{cases}$$

Especialização em Instâncias

Função factorial

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n > 0 \end{cases}$$

Função fact

```
(defgeneric fact (n))
```

Especialização em Instâncias

Função factorial

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n > 0 \end{cases}$$

Função fact

```
(defgeneric fact (n))

(defmethod fact ((n integer)) ;;there is no class for n > 0
  (* n (fact (1- n))))
```

Especialização em Instâncias

Função factorial

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n > 0 \end{cases}$$

Função fact

```
(defgeneric fact (n))

(defmethod fact ((n integer)) ;;there is no class for n > 0
  (* n (fact (1- n))))

(defmethod fact ((n (eql 0))) ;;but we can specialize on 0
  1)
```

Especialização em Instâncias

Função factorial

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n > 0 \end{cases}$$

Função fact

```
(defgeneric fact (n))

(defmethod fact ((n integer)) ;;there is no class for n > 0
  (* n (fact (1- n))))

(defmethod fact ((n (eql 0))) ;;but we can specialize on 0
  1)

> (fact 5)
```

Especialização em Instâncias

Função factorial

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n > 0 \end{cases}$$

Função fact

```
(defgeneric fact (n))

(defmethod fact ((n integer)) ;;there is no class for n > 0
  (* n (fact (1- n))))

(defmethod fact ((n (eql 0))) ;;but we can specialize on 0
  1)

> (fact 5)
120
```

Especialização em Instâncias

Função foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{se } x = 5! \\ 0 & \text{caso contrário} \end{cases}$$

Especialização em Instâncias

Função foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{se } x = 5! \\ 0 & \text{caso contrário} \end{cases}$$

Função foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)
```

Especialização em Instâncias

Função foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{se } x = 5! \\ 0 & \text{caso contrário} \end{cases}$$

Função foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)

(defmethod foobar ((x t))
  0)
```

Especialização em Instâncias

Função foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{se } x = 5! \\ 0 & \text{caso contrário} \end{cases}$$

Função foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)

(defmethod foobar ((x t))
  0)

> (foobar 34)
```

Especialização em Instâncias

Função foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{se } x = 5! \\ 0 & \text{caso contrário} \end{cases}$$

Função foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)

(defmethod foobar ((x t))
  0)

> (foobar 34)
0
```

Especialização em Instâncias

Função foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{se } x = 5! \\ 0 & \text{caso contrário} \end{cases}$$

Função `foobar`

```
(defmethod foobar ((x (eql (fact 5))))
  1)

(defmethod foobar ((x t))
  0)

> (foobar 34)
0
> (foobar (fact 5))
```

Especialização em Instâncias

Função foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{se } x = 5! \\ 0 & \text{caso contrário} \end{cases}$$

Função foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)

(defmethod foobar ((x t))
  0)

> (foobar 34)
0
> (foobar (fact 5))
1
```

Especialização em Instâncias

Função foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{se } x = 5! \\ 0 & \text{caso contrário} \end{cases}$$

Função foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)

(defmethod foobar ((x t))
  0)

> (foobar 34)
0
> (foobar (fact 5))
1
> (foobar 120)
```

Especialização em Instâncias

Função foobar

$$\text{foobar}(x) = \begin{cases} 1 & \text{se } x = 5! \\ 0 & \text{caso contrário} \end{cases}$$

Função foobar

```
(defmethod foobar ((x (eql (fact 5))))
  1)

(defmethod foobar ((x t))
  0)

> (foobar 34)
0
> (foobar (fact 5))
1
> (foobar 120)
1
```

Especialização em Instâncias

Função Fibonacci

$$\text{fib}(n) = \begin{cases} 0 & \text{se } n = 0; \\ 1 & \text{se } n = 1; \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{caso contrário} \end{cases}$$

Especialização em Instâncias

Função Fibonacci

$$\text{fib}(n) = \begin{cases} 0 & \text{se } n = 0; \\ 1 & \text{se } n = 1; \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{caso contrário} \end{cases}$$

Função fib

```
(defgeneric fib (n))

(defmethod fib ((n (eql 0)))
  0)

(defmethod fib ((n (eql 1)))
  1)

(defmethod fib ((n number))
  (+ (fib (- n 1)) (fib (- n 2))))
```

Combinação de Métodos

Exemplo

```
> (time (fib 40))
; real time  22,612 msec
102334155
```

Combinação de Métodos

Exemplo

```
> (time (fib 40))
; real time  22,612 msec
102334155
```

Memoization

Combinação de Métodos

Exemplo

```
> (time (fib 40))
; real time  22,612 msec
102334155
```

Memoization

```
(let ((cached-results (make-hash-table)))
```

Combinação de Métodos

Exemplo

```
> (time (fib 40))
; real time  22,612 msec
102334155
```

Memoization

```
(let ((cached-results (make-hash-table)))
  (defmethod fib :around ((n number))
```

Combinação de Métodos

Exemplo

```
> (time (fib 40))
; real time  22,612 msec
102334155
```

Memoization

```
(let ((cached-results (make-hash-table)))
  (defmethod fib :around ((n number))
    (or (gethash n cached-results)
```

Combinação de Métodos

Exemplo

```
> (time (fib 40))
; real time  22,612 msec
102334155
```

Memoization

```
(let ((cached-results (make-hash-table)))
  (defmethod fib :around ((n number))
    (or (gethash n cached-results)
      (setf (gethash n cached-results)
            (call-next-method)))))
```

Combinação de Métodos

Exemplo

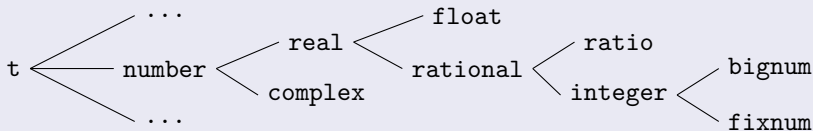
```
> (time (fib 40))
; real time  22,612 msec
102334155
```

Memoization

```
(let ((cached-results (make-hash-table)))
  (defmethod fib :around ((n number))
    (or (gethash n cached-results)
      (setf (gethash n cached-results)
            (call-next-method)))))
```

Exemplo

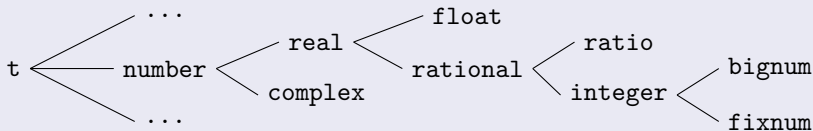
```
CL-USER> (time (fib 40))
; real time  10 msec
102334155
```

Combinação de Métodos

Hierarquia de Tipos Numéricos

```
      ...                        float
t --- number --- real    --- rational --- ratio
      ...        complex                  integer --- bignum
                                                      fixnum
```

Função explain

```
(defgeneric explain (entity)
  (:method ((entity   fixnum)) (format t "~S is a fixnum" entity))
  (:method ((entity rational)) (format t "~S is a rational" entity))
  (:method ((entity   string)) (format t "~S is a string" entity)))
```

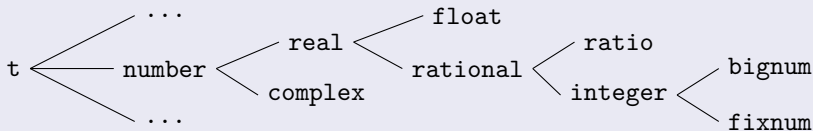
Combinação de Métodos

Hierarquia de Tipos Numéricos

Função explain

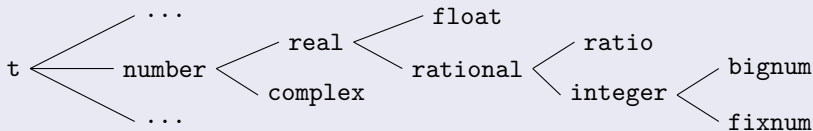
```
(defgeneric explain (entity)
  (:method ((entity   fixnum)) (format t "~S is a fixnum" entity))
  (:method ((entity rational)) (format t "~S is a rational" entity))
  (:method ((entity   string)) (format t "~S is a string" entity)))

> (explain 123)
```

Combinação de Métodos

Hierarquia de Tipos Numéricos

- t
  - ...
  - number
    - real
      - float
      - rational
        - ratio
        - integer
          - bignum
          - fixnum
    - complex
  - ...

Função explain

```
(defgeneric explain (entity)
  (:method ((entity   fixnum)) (format t "~S is a fixnum" entity))
  (:method ((entity rational)) (format t "~S is a rational" entity))
  (:method ((entity   string)) (format t "~S is a string" entity)))

> (explain 123)
123 is a fixnum
```

Combinação de Métodos

Hierarquia de Tipos Numéricos

Função explain

```
(defgeneric explain (entity)
  (:method ((entity   fixnum)) (format t "~S is a fixnum" entity))
  (:method ((entity rational)) (format t "~S is a rational" entity))
  (:method ((entity   string)) (format t "~S is a string" entity)))

> (explain 123)
123 is a fixnum
> (explain "Hi")
```

Combinação de Métodos

Hierarquia de Tipos Numéricos

- t
  - ...
  - number
    - real
      - float
      - rational
        - ratio
        - integer
          - bignum
          - fixnum
    - complex
  - ...

Função explain

```
(defgeneric explain (entity)
  (:method ((entity   fixnum)) (format t "~S is a fixnum" entity))
  (:method ((entity rational)) (format t "~S is a rational" entity))
  (:method ((entity   string)) (format t "~S is a string" entity)))

> (explain 123)
123 is a fixnum
> (explain "Hi")
"Hi" is a string
```

Combinação de Métodos

Hierarquia de Tipos Numéricos

```
        ...                       float
                  real <                     ratio
t  <  number <                  rational <             bignum
                  complex                    integer <
        ...                                            fixnum
```

Função explain

```
(defgeneric explain (entity)
  (:method ((entity   fixnum)) (format t "~S is a fixnum" entity))
  (:method ((entity rational)) (format t "~S is a rational" entity))
  (:method ((entity   string)) (format t "~S is a string" entity)))

> (explain 123)
123 is a fixnum
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
```

Combinação de Métodos

Hierarquia de Tipos Numéricos

- t
  - ...
  - number
    - real
      - float
      - rational
        - ratio
        - integer
          - bignum
          - fixnum
    - complex
  - ...

Função explain

```
(defgeneric explain (entity)
  (:method ((entity   fixnum)) (format t "~S is a fixnum" entity))
  (:method ((entity rational)) (format t "~S is a rational" entity))
  (:method ((entity   string)) (format t "~S is a string" entity)))

> (explain 123)
123 is a fixnum
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational
```

Combinação de Métodos

Função explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))
```

Combinação de Métodos

Função explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
```

Combinação de Métodos

Função explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
```

Combinação de Métodos

Função explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
```

Combinação de Métodos

Função explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
```

Combinação de Métodos

Função explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
```

Combinação de Métodos

Função explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational
```

Combinação de Métodos

Função explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational

(defmethod explain :before ((entity number))
  (format t "The number "))
```

Combinação de Métodos

Função explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational

(defmethod explain :before ((entity number))
  (format t "The number "))

> (explain 123)
```

Combinação de Métodos

Função explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational

(defmethod explain :before ((entity number))
  (format t "The number "))

> (explain 123)
The number 123 is a fixnum (in binary, is 1111011)
```

Combinação de Métodos

Função explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational

(defmethod explain :before ((entity number))
  (format t "The number "))

> (explain 123)
The number 123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
```

Combinação de Métodos

Função explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational

(defmethod explain :before ((entity number))
  (format t "The number "))

> (explain 123)
The number 123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
```

Combinação de Métodos

Função explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational

(defmethod explain :before ((entity number))
  (format t "The number "))

> (explain 123)
The number 123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
```

Combinação de Métodos

Função explain

```
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

> (explain 123)
123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
1/3 is a rational

(defmethod explain :before ((entity number))
  (format t "The number "))

> (explain 123)
The number 123 is a fixnum (in binary, is 1111011)
> (explain "Hi")
"Hi" is a string
> (explain 1/3)
The number 1/3 is a rational
```

Funções Genéricas

Aplicação de uma função genérica a argumentos

1. Determinar o **método efectivo**.
2. Se não existe, invocar a função genérica `no-applicable-method` usando, como argumentos, a função genérica em questão juntamente com os seus argumentos.
3. Se existe, invocar o **método efectivo** com os mesmo argumentos da função genérica.

Determinar o método efectivo

1. Seleccionar os **métodos aplicáveis**.
2. **Ordenar** os métodos por precedência, do mais específico para o menos específico.
3. **Combinar** os métodos aplicáveis.

Funções Genéricas

Método aplicável

- Dada uma função genérica e uma lista de argumentos obrigatórios a_0, ..., a_n, um método aplicável é um método dessa função genérica cujos especializadores de parâmetros p_0, ..., p_n são satisfeitos pelos argumentos correspondentes.
- Um especializador de parâmetro p_i é satisfeito pelo argumento correspondente a_i se (typep a_i 'p_i).

Métodos aplicáveis a (explain 123)

```
(defmethod explain ((entity fixnum))
  (format t "~S is a fixnum" entity))
(defmethod explain ((entity rational))
  (format t "~S is a rational" entity))
(defmethod explain :before ((entity number))
  (format t "The number "))
(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))
```

Funções Genéricas

Qualificadores

- Cada método pode ter zero ou mais *qualificadores*.
- Um qualificador pode ser qualquer objecto excepto uma lista (para distinguir os qualificadores da lista de parâmetros).
- A **combinação *standard* de métodos** distingue:

  Métodos Primários: métodos não-qualificados.
  Métodos Auxiliares: métodos qualificados com os símbolos `:before`, `:after` e `:around`.
- Outras combinações de métodos podem distinguir outros tipos de métodos.
- É possível definir novas combinações de métodos.

Funções Genéricas

Métodos primários aplicáveis a (explain 123)

```
(defmethod explain ((entity fixnum))
  (format t "~S is a fixnum" entity))

(defmethod explain ((entity rational))
  (format t "~S is a rational" entity))
```

Métodos auxiliares aplicáveis a (explain 123)

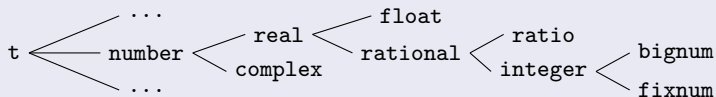
```
(defmethod explain :before ((entity number))
  (format t "The number "))

(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))
```

Funções Genéricas

Ordenação de Métodos

- Ordena os métodos do mais específico para o menos específico comparando-os dois a dois.
- Dados dois métodos aplicáveis:
    1. Compara-se os pares de especializadores de parâmetro por ordem (por omissão, da esquerda para a direita).
    2. Quando os pares de especializadores diferem, o método com maior precedência é aquele cujo especializador de parâmetro aparece primeiro na **lista de precedência de classes** do argumento correspondente.
    3. Se um dos especializadores é de instância (`(eql` *`objecto`*`)`) então esse método tem precedência sobre o outro.
    4. Se todos os especializadores são iguais, os qualificadores dos métodos são necessariamente diferentes e qualquer dos métodos pode ter precedência sobre o outro.

Funções Genéricas

Lista de Precedência de Classes de 123

`fixnum`, `integer`, `rational`, `real`, `number`, `t`

Ordenação de Métodos aplicáveis a (explain 123)

```
(defmethod explain ((entity fixnum))
  (format t "~S is a fixnum" entity))

(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))

(defmethod explain ((entity rational))
  (format t "~S is a rational" entity))

(defmethod explain :before ((entity number))
  (format t "The number "))
```

Funções Genéricas

Combinação de Métodos

- Realizada após a seleccção e ordenação de métodos aplicáveis.
- Responsável por produzir o **método efectivo** que vai ser aplicado aos argumentos da função genérica.
- Existem várias formas pré-definidas de combinar os métodos (denominados *tipos de combinação*):

  Simples `append`, `nconc`, `list`, `progn`, `max`, `min`, `+`, `and`, `or`
  Implica especificar o tipo de combinação na função genérica e em todos os métodos dessa função.

  Standard `standard`
  É empregue por omissão se nada for especificado na função genérica. Implicitamente usado quando não se especifica a função genérica.

Funções Genéricas

Combinação **Standard** de Métodos

- Os métodos primários definem o comportamento fundamental. Apenas o mais específico é executado mas pode executar os restantes através de `call-next-method`.
- Os métodos auxiliares modificam o comportamento dos métodos primários:

  `:before` Métodos que são executados *antes* dos métodos primários.

  `:after` Métodos que são executados *depois* dos métodos primários.

  `:around` Código que é executado *no lugar* dos métodos aplicáveis (incluíndo os métodos primários e outros métodos auxiliares) mas que os pode executar através de `call-next-method`.

Funções Genéricas

Combinação **Standard** de Métodos

Se não existem métodos `:around`:

1. Todos os métodos `:before` são executados, do mais específico para o menos específico, ignorando-se os seus valores.
2. O método primário mais específico é executado.
   - Se esse método invoca `call-next-method`, o próximo método mais específico é executado e os seus valores são devolvidos ao método que invocou `call-next-method`.
   - Os valores devolvidos pelo método primário mais específico são os valores devolvidos pela invocação da função genérica.
3. Todos os métodos `:after` são executados, do menos específico para o mais específico, ignorando-se os seus valores.

Funções Genéricas

Combinação **Standard** de Métodos

Se existem métodos `:around`, o mais específico é executado. Se esse método invoca `call-next-method`:

1. Se existir, o próximo método `:around` mais específico é executado e os seus valores são devolvidos ao método que invocou `call-next-method`.
2. Se não existir outro método `:around`:
   1. Todos os métodos `:before` são executados, do mais específico para o menos específico, ignorando-se os seus valores.
   2. O método primário mais específico é executado. Se esse método invoca `call-next-method`, o próximo método mais específico é executado e os seus valores são devolvidos ao método que invocou `call-next-method`.
   3. Todos os métodos `:after` são executados, do menos específico para o mais específico, ignorando-se os seus valores.

Funções Genéricas

Combinação **Standard** de Métodos

- A invocação da função `call-next-method` pode ser feita:
  - Sem argumentos: implica usar os mesmos argumentos que foram usados na invocação do método.
  - Com argumentos: usa novos argumentos mas os novos argumentos devem implicar a mesma sequência ordenada de métodos aplicáveis que foi usada para os argumentos originais.
- Se, quando se invoca a função `call-next-method`, não existir mais nenhum método aplicável, é invocada automaticamente a função genérica `no-next-method` usando, como argumentos:
  - A função genérica a que pertence o método que invocou `call-next-method`.
  - O método que invocou `call-next-method`.
  - Os argumentos que foram passados ao `call-next-method`.
- A função `next-method-p` testa se existe mais algum método.

Funções Genéricas

Métodos aplicáveis a (explain 123) segundo a combinação standard

```
(defmethod explain :before ((entity number))
  (format t "The number "))

(defmethod explain ((entity fixnum))
  (format t "~S is a fixnum" entity))

(defmethod explain ((entity rational))
  (format t "~S is a rational" entity))

(defmethod explain :after ((entity integer))
  (format t " (in binary, is ~B)" entity))
```

O Método Efectivo aplicado a (explain 123) (simplificado)

```
(lambda (entity)
  (format t "The number ")
  (format t "~S is a fixnum" entity)
  (format t " (in binary, is ~B)" entity))
```

Funções Genéricas

Combinação **Simples** de Métodos

A **combinação *simples* de métodos** distingue:

Métodos Primários: métodos qualificados com o tipo de combinação (`append`, `nconc`, `list`, `progn`, `max`, `min`, `+`, `and`, `or`).

Métodos Auxiliares: métodos qualificados com o símbolo `:around`.

Se não existem métodos `:around`:

1. O método efectivo é construído combinando o operador indicado pelo tipo de combinação com as invocações de todos os métodos primários pela sua ordem de especificidade (ou pela ordem inversa, se tal for indicado na função genérica).

Funções Genéricas

Combinação **Simples** de Métodos

Se existem métodos `:around`, o mais específico é executado. Se esse método invoca `call-next-method`:

1. Se existir, o próximo método `:around` mais específico é executado e os seus valores são devolvidos ao método que invocou `call-next-method`.
2. Se não existir outro método `:around`:
    1. O método efectivo é construído combinando o operador indicado pelo tipo de combinação com as invocações de todos os métodos primários pela sua ordem de especificidade (ou pela ordem inversa, se tal for indicado na função genérica).

Funções Genéricas

Combinação **Simples** de Métodos

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))

(defmethod what-are-you? list ((obj fixnum))
  "I am a FIXNUM")

(defmethod what-are-you? list ((obj float))
  "I am a FLOAT")

(defmethod what-are-you? list ((obj number))
  "I am a NUMBER")
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))

(defmethod what-are-you? list ((obj fixnum))
  "I am a FIXNUM")

(defmethod what-are-you? list ((obj float))
  "I am a FLOAT")

(defmethod what-are-you? list ((obj number))
  "I am a NUMBER")

> (what-are-you? 123)
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))

(defmethod what-are-you? list ((obj fixnum))
  "I am a FIXNUM")

(defmethod what-are-you? list ((obj float))
  "I am a FLOAT")

(defmethod what-are-you? list ((obj number))
  "I am a NUMBER")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))

(defmethod what-are-you? list ((obj fixnum))
  "I am a FIXNUM")

(defmethod what-are-you? list ((obj float))
  "I am a FLOAT")

(defmethod what-are-you? list ((obj number))
  "I am a NUMBER")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))

(defmethod what-are-you? list ((obj fixnum))
  "I am a FIXNUM")

(defmethod what-are-you? list ((obj float))
  "I am a FLOAT")

(defmethod what-are-you? list ((obj number))
  "I am a NUMBER")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defgeneric what-are-you? (obj)
  (:method-combination list :most-specific-last))

(defmethod what-are-you? list ((obj fixnum))
  "I am a FIXNUM")

(defmethod what-are-you? list ((obj float))
  "I am a FLOAT")

(defmethod what-are-you? list ((obj number))
  "I am a NUMBER")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
("I am a NUMBER")
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
("I am a NUMBER" "I am a RATIO")
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
("I am a NUMBER" "I am a RATIO")

(defmethod what-are-you? list ((obj (eql 1)))
  "I am THE SPECIAL ONE")
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
("I am a NUMBER" "I am a RATIO")

(defmethod what-are-you? list ((obj (eql 1)))
  "I am THE SPECIAL ONE")

> (what-are-you? 0)
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
("I am a NUMBER" "I am a RATIO")

(defmethod what-are-you? list ((obj (eql 1)))
  "I am THE SPECIAL ONE")

> (what-are-you? 0)
("I am a NUMBER" "I am a FIXNUM")
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj ratio))
  "I am a RATIO")

> (what-are-you? 123)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1.23)
("I am a NUMBER" "I am a FLOAT")

> (what-are-you? 1/3)
("I am a NUMBER" "I am a RATIO")

(defmethod what-are-you? list ((obj (eql 1)))
  "I am THE SPECIAL ONE")

> (what-are-you? 0)
("I am a NUMBER" "I am a FIXNUM")

> (what-are-you? 1)
("I am a NUMBER" "I am a FIXNUM" "I am THE SPECIAL ONE")
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj null))
  "I am a NULL")

(defmethod what-are-you? list ((obj symbol))
  "I am a SYMBOL")

(defmethod what-are-you? list ((obj list))
  "I am a LIST")
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj null))
  "I am a NULL")

(defmethod what-are-you? list ((obj symbol))
  "I am a SYMBOL")

(defmethod what-are-you? list ((obj list))
  "I am a LIST")

> (what-are-you? 'hi)
("I am a SYMBOL")
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj null))
  "I am a NULL")

(defmethod what-are-you? list ((obj symbol))
  "I am a SYMBOL")

(defmethod what-are-you? list ((obj list))
  "I am a LIST")

> (what-are-you? 'hi)
("I am a SYMBOL")

> (what-are-you? '(1 2 3))
("I am a LIST")
```

Funções Genéricas

Combinação **Simples** de Métodos

```
(defmethod what-are-you? list ((obj null))
  "I am a NULL")

(defmethod what-are-you? list ((obj symbol))
  "I am a SYMBOL")

(defmethod what-are-you? list ((obj list))
  "I am a LIST")

> (what-are-you? 'hi)
("I am a SYMBOL")

> (what-are-you? '(1 2 3))
("I am a LIST")

> (what-are-you? '())
("I am a LIST" "I am a SYMBOL" "I am a NULL")
```

Funções Genéricas

Combinação de Métodos definida pelo utilizador

```
(define-method-combination list ()
         ((methods (list)))
   `(list ,@(mapcar #'(lambda (method)
                       `(call-method ,method))
                  methods)))
```

Definição

Funções Genéricas

Combinação de Métodos definida pelo utilizador

```
(define-method-combination list ()
         ((methods (list)))
   `(list ,@(mapcar #'(lambda (method)
                        `(call-method ,method))
                  methods)))
```

Definição

- Nome da combinação de método.

Funções Genéricas

Combinação de Métodos definida pelo utilizador

```
(define-method-combination list ()
         ((methods (list)))
   `(list ,@(mapcar #'(lambda (method)
                        `(call-method ,method))
                  methods)))
```

Definição

- Nome da combinação de método.
- Parâmetros da combinação de método (por exemplo, a ordenação dos métodos).

Funções Genéricas

Combinação de Métodos definida pelo utilizador

```
(define-method-combination list ()
         ((methods (list)))
   `(list ,@(mapcar #'(lambda (method)
                        `(call-method ,method))
                  methods)))
```

Definição

- Nome da combinação de método.
- Parâmetros da combinação de método (por exemplo, a ordenação dos métodos).
- Variável local para conter os métodos cujos qualificadores...

Funções Genéricas

Combinação de Métodos definida pelo utilizador

```
(define-method-combination list ()
         ((methods (list)))
   `(list ,@(mapcar #'(lambda (method)
                        `(call-method ,method))
                  methods)))
```

Definição

- Nome da combinação de método.
- Parâmetros da combinação de método (por exemplo, a ordenação dos métodos).
- Variável local para conter os métodos cujos qualificadores...
- ...satisfazem este padrão

Funções Genéricas

Combinação de Métodos definida pelo utilizador

```
(define-method-combination list ()
         ((methods (list)))
   `(list ,@(mapcar #'(lambda (method)
                          `(call-method ,method))
                  methods)))
```

Definição

- Nome da combinação de método.
- Parâmetros da combinação de método (por exemplo, a ordenação dos métodos).
- Variável local para conter os métodos cujos qualificadores...
- ...satisfazem este padrão
- Invocação de cada método no método efectivo

Funções Genéricas

Combinação **Standard** de Métodos

```
(define-method-combination standard ()
         ((around (:around))
          (before (:before))
          (primary () :required t)
          (after (:after)))
   (flet ((call-methods (methods)
            (mapcar #'(lambda (method)
                        `(call-method ,method))
                    methods)))
     (let ((form (if (or before after (rest primary))
                     `(multiple-value-prog1
                        (progn ,@(call-methods before)
                               (call-method ,(first primary)
                                            ,(rest primary)))
                        ,@(call-methods (reverse after)))
                     `(call-method ,(first primary)))))
       (if around
           `(call-method ,(first around)
                         (,@(rest around)
                          (make-method ,form)))
           form))))
```

Classes

Definição de classes com defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
        :reader foo-slot1
        :writer set-foo-slot1)
   (slot2 :type string
        :initarg :slot2
        :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))
```

Classes

Definição de classes com defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
         :reader foo-slot1
         :writer set-foo-slot1)
   (slot2 :type string
         :initarg :slot2
         :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))
```

Classes

Definição de classes com defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))
```

Classes

Definição de classes com defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
        :reader foo-slot1
        :writer set-foo-slot1)
   (slot2 :type string
        :initarg :slot2
        :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))
```

Classes

Definição de classes com defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))
```

Classes

Definição de classes com defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
        :reader foo-slot1
        :writer set-foo-slot1)
   (slot2 :type string
        :initarg :slot2
        :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))

(defmethod foo-slot1 ((obj foo))
  (slot-value obj 'slot1))
```

Classes

Definição de classes com defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
        :reader foo-slot1
        :writer set-foo-slot1)
   (slot2 :type string
        :initarg :slot2
        :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))

(defmethod foo-slot1 ((obj foo))
  (slot-value obj 'slot1))

(defmethod set-foo-slot1 ((obj foo) new-value)
  (setf (slot-value obj 'slot1) new-value))
```

Classes

Definição de classes com defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))

(defmethod foo-slot1 ((obj foo))
  (slot-value obj 'slot1))

(defmethod set-foo-slot1 ((obj foo) new-value)
  (setf (slot-value obj 'slot1) new-value))
```

Classes

Definição de classes com defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))

(defmethod foo-slot1 ((obj foo))
  (slot-value obj 'slot1))

(defmethod set-foo-slot1 ((obj foo) new-value)
  (setf (slot-value obj 'slot1) new-value))
```

Classes

Definição de classes com defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
         :reader foo-slot1
         :writer set-foo-slot1)
   (slot2 :type string
         :initarg :slot2
         :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))

(defmethod foo-slot1 ((obj foo))
  (slot-value obj 'slot1))

(defmethod set-foo-slot1 ((obj foo) new-value)
  (setf (slot-value obj 'slot1) new-value))

(defmethod foo-slot2 ((obj foo))
  (slot-value obj 'slot2))

(defmethod (setf foo-slot2) (new-value (obj foo))
  (setf (slot-value obj 'slot2) new-value))
```

Classes

Definição de classes com defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
          :reader foo-slot1
          :writer set-foo-slot1)
   (slot2 :type string
          :initarg :slot2
          :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))

(defmethod foo-slot1 ((obj foo))
  (slot-value obj 'slot1))

(defmethod set-foo-slot1 ((obj foo) new-value)
  (setf (slot-value obj 'slot1) new-value))

(defmethod foo-slot2 ((obj foo))
  (slot-value obj 'slot2))

(defmethod (setf foo-slot2) (new-value (obj foo))
  (setf (slot-value obj 'slot2) new-value))
```

Classes

Definição de classes com defclass

```
(defclass foo (bar baz)
  ((slot1 :initform (fact 5)
        :reader foo-slot1
        :writer set-foo-slot1)
   (slot2 :type string
        :initarg :slot2
        :accessor foo-slot2)
   (slot3 :allocation :class))
  (:default-initargs :slot2 "hi there"))

(defmethod foo-slot1 ((obj foo))
  (slot-value obj 'slot1))

(defmethod set-foo-slot1 ((obj foo) new-value)
  (setf (slot-value obj 'slot1) new-value))

(defmethod foo-slot2 ((obj foo))
  (slot-value obj 'slot2))

(defmethod (setf foo-slot2) (new-value (obj foo))
  (setf (slot-value obj 'slot2) new-value))
```

CLOS-Common Lisp Object System

Classes

```
(defclass shape ()
  ())

(defclass device ()
  ())
```

CLOS-Common Lisp Object System

Classes

```
(defclass shape ()
  ())

(defclass device ()
  ())

(defgeneric draw (shape device))

(defmethod draw ((s shape) (d device))
  (format t "draw what where?~%"))
```

CLOS-Common Lisp Object System

Classes

```
(defclass shape ()
  ())

(defclass device ()
  ())

(defgeneric draw (shape device))

(defmethod draw ((s shape) (d device))
  (format t "draw what where?~%"))

(defclass line (shape)
  ())

(defclass circle (shape)
  ())
```

CLOS-Common Lisp Object System

Classes

```
(defclass shape ()
  ())

(defclass device ()
  ())

(defgeneric draw (shape device))

(defmethod draw ((s shape) (d device))
  (format t "draw what where?~%"))

(defclass line (shape)
  ())

(defclass circle (shape)
  ())

(defclass screen (device)
  ())

(defclass printer (device)
  ())
```

CLOS-Common Lisp Object System

Despacho Múltiplo

```
(defmethod draw ((s line) (d device))
  (format t "draw a line where?~%"))

(defmethod draw ((s circle) (d device))
  (format t "draw a circle where?~%"))
```

CLOS-Common Lisp Object System

Despacho Múltiplo

```
(defmethod draw ((s line) (d device))
  (format t "draw a line where?~%"))

(defmethod draw ((s circle) (d device))
  (format t "draw a circle where?~%"))

(defmethod draw ((s shape) (d screen))
  (format t "draw what on screen?~%"))

(defmethod draw ((s shape) (d printer))
  (format t "draw what on printer?~%"))
```

CLOS-Common Lisp Object System

Despacho Múltiplo

```
(defmethod draw ((s line) (d screen))
  (format t "drawing a line on screen!~%"))

(defmethod draw ((s circle) (d screen))
  (format t "drawing a circle on screen!~%"))

(defmethod draw ((s line) (d printer))
  (format t "drawing a line on printer!~%"))

(defmethod draw ((s circle) (d printer))
  (format t "drawing a circle on printer!~%"))
```

CLOS-Common Lisp Object System

Despacho Múltiplo

```
(let ((shapes (list (make-instance 'line)
                    (make-instance 'circle)))
```

CLOS-Common Lisp Object System

Despacho Múltiplo

```
(let ((shapes (list (make-instance 'line)
                    (make-instance 'circle)))
      (devices (list (make-instance 'screen)
                     (make-instance 'printer))))
```

CLOS-Common Lisp Object System

Despacho Múltiplo

```
(let ((shapes (list (make-instance 'line)
                    (make-instance 'circle)))
      (devices (list (make-instance 'screen)
                     (make-instance 'printer))))
  (dolist (device devices)
    (dolist (shape shapes)
      (draw shape device))))
```

CLOS-Common Lisp Object System

Despacho Múltiplo

```
(let ((shapes (list (make-instance 'line)
                    (make-instance 'circle)))
      (devices (list (make-instance 'screen)
                     (make-instance 'printer))))
  (dolist (device devices)
    (dolist (shape shapes)
      (draw shape device))))

drawing a line on screen!
drawing a circle on screen!
drawing a line on printer!
drawing a circle on printer!
```

CLOS-Common Lisp Object System

Slots

```
(defclass 2d-position ()
  ((x :initarg :x)
   (y :initarg :y)))
```

CLOS-Common Lisp Object System

Slots

```
(defclass 2d-position ()
  ((x :initarg :x)
   (y :initarg :y)))

(defclass line (shape)
  ((origin :initarg :origin :accessor line-origin)
   (end :initarg :end :accessor line-end)))

(defclass circle (shape)
  ((center :initarg :center :accessor circle-center)
   (radius :initarg :radius :accessor circle-radius :initform 1)))
```

CLOS-Common Lisp Object System

Slots

```
(defclass 2d-position ()
  ((x :initarg :x)
   (y :initarg :y)))

(defclass line (shape)
  ((origin :initarg :origin :accessor line-origin)
   (end :initarg :end :accessor line-end)))

(defclass circle (shape)
  ((center :initarg :center :accessor circle-center)
   (radius :initarg :radius :accessor circle-radius :initform 1)))

> (make-instance 'circle
    :center (make-instance '2d-position :x 10 :y 30)
    :radius 5)
```

CLOS-Common Lisp Object System

Slots

```
(defclass 2d-position ()
  ((x :initarg :x)
   (y :initarg :y)))

(defclass line (shape)
  ((origin :initarg :origin :accessor line-origin)
   (end :initarg :end :accessor line-end)))

(defclass circle (shape)
  ((center :initarg :center :accessor circle-center)
   (radius :initarg :radius :accessor circle-radius :initform 1)))

> (make-instance 'circle
    :center (make-instance '2d-position :x 10 :y 30)
    :radius 5)
#<CIRCLE @ #x71641c1a>
```

CLOS-Common Lisp Object System

Slots

```
(defclass 2d-position ()
  ((x :initarg :x)
   (y :initarg :y)))

(defclass line (shape)
  ((origin :initarg :origin :accessor line-origin)
   (end :initarg :end :accessor line-end)))

(defclass circle (shape)
  ((center :initarg :center :accessor circle-center)
   (radius :initarg :radius :accessor circle-radius :initform 1)))

> (make-instance 'circle
    :center (make-instance '2d-position :x 10 :y 30)
    :radius 5)
#<CIRCLE @ #x71641c1a>

> (circle-radius (make-instance 'circle))
```

CLOS-Common Lisp Object System

Slots

```
(defclass 2d-position ()
  ((x :initarg :x)
   (y :initarg :y)))

(defclass line (shape)
  ((origin :initarg :origin :accessor line-origin)
   (end :initarg :end :accessor line-end)))

(defclass circle (shape)
  ((center :initarg :center :accessor circle-center)
   (radius :initarg :radius :accessor circle-radius :initform 1)))

> (make-instance 'circle
    :center (make-instance '2d-position :x 10 :y 30)
    :radius 5)
#<CIRCLE @ #x71641c1a>

> (circle-radius (make-instance 'circle))
1
```

CLOS-Common Lisp Object System

Mixins

```
(defclass color-mixin ()
  ((color :initarg :color :accessor color)))
```

CLOS-Common Lisp Object System

Mixins

```
(defclass color-mixin ()
  ((color :initarg :color :accessor color)))

(defmethod draw :around ((s color-mixin) (d device))
  (let ((previous-color (color d)))
    (setf (color d) (color s))
    (unwind-protect
       (call-next-method)
      (setf (color d) previous-color))))
```

CLOS-Common Lisp Object System

Mixins

```
(defclass color-mixin ()
  ((color :initarg :color :accessor color)))

(defmethod draw :around ((s color-mixin) (d device))
  (let ((previous-color (color d)))
    (setf (color d) (color s))
    (unwind-protect
       (call-next-method)
      (setf (color d) previous-color))))

(defclass colored-line (color-mixin line)
  ())

(defclass colored-circle (color-mixin circle)
  ())
```

CLOS-Common Lisp Object System

Mixins

```
(defclass colored-printer (printer)
  ((ink :initform :black :accessor color)))

(defmethod (setf color) :before (color (d colored-printer))
  (format t "changing printer ink color to ~A~%" color))
```

CLOS-Common Lisp Object System

Mixins

```
(defclass colored-printer (printer)
  ((ink :initform :black :accessor color)))

(defmethod (setf color) :before (color (d colored-printer))
  (format t "changing printer ink color to ~A~%" color))

(let ((shapes (list (make-instance 'line)
                    (make-instance 'colored-circle :color :red)
                    (make-instance 'colored-line :color :blue)))
      (printer (make-instance 'colored-printer)))
  (dolist (shape shapes)
    (draw shape printer)))
```

CLOS-Common Lisp Object System

Mixins

```
(defclass colored-printer (printer)
  ((ink :initform :black :accessor color)))

(defmethod (setf color) :before (color (d colored-printer))
  (format t "changing printer ink color to ~A~%" color))

(let ((shapes (list (make-instance 'line)
                    (make-instance 'colored-circle :color :red)
                    (make-instance 'colored-line :color :blue)))
      (printer (make-instance 'colored-printer)))
  (dolist (shape shapes)
    (draw shape printer)))

drawing a line on printer!
```

CLOS-Common Lisp Object System

Mixins

```
(defclass colored-printer (printer)
  ((ink :initform :black :accessor color)))

(defmethod (setf color) :before (color (d colored-printer))
  (format t "changing printer ink color to ~A~%" color))

(let ((shapes (list (make-instance 'line)
                    (make-instance 'colored-circle :color :red)
                    (make-instance 'colored-line :color :blue)))
      (printer (make-instance 'colored-printer)))
  (dolist (shape shapes)
    (draw shape printer)))

drawing a line on printer!
changing printer ink color to RED
drawing a circle on printer!
changing printer ink color to BLACK
```

CLOS-Common Lisp Object System

Mixins

```
(defclass colored-printer (printer)
  ((ink :initform :black :accessor color)))

(defmethod (setf color) :before (color (d colored-printer))
  (format t "changing printer ink color to ~A~%" color))

(let ((shapes (list (make-instance 'line)
                    (make-instance 'colored-circle :color :red)
                    (make-instance 'colored-line :color :blue)))
      (printer (make-instance 'colored-printer)))
  (dolist (shape shapes)
    (draw shape printer)))

drawing a line on printer!
changing printer ink color to RED
drawing a circle on printer!
changing printer ink color to BLACK
changing printer ink color to BLUE
drawing a line on printer!
changing printer ink color to BLACK
```

Classes

Herança de Classes

- A classe C_1 é uma **subclasse directa** da classe C_2 (a classe C_2 é uma **superclasse directa** da classe C_1) se C_1, na sua definição, explicitamente designa C_2 na lista de superclasses.
- A classe C_1 é uma **subclasse** da classe C_n (a classe C_n é uma **superclasse** da classe C_1) se existir uma sequência de classes $C_2, \ldots, C_{n-1}$ tais que C_i é uma **subclasse directa** de C_{i+1}, $0 < i < n$.
- A **lista de precedências da classe** C é uma ordenação total do conjunto contendo C e todas as suas superclasses, da mais específica para a menos específica.
- A ordenação da **lista de precedências da classe** C é consistente com a ordenação local de **superclasses directas** presente na definição de C.

Classes

Lista de Precedências de Classes

Flavors Profundidade primeiro, da esquerda para a direita, sem os últimos duplicados (`standard-object` e `t` adicionados no fim).

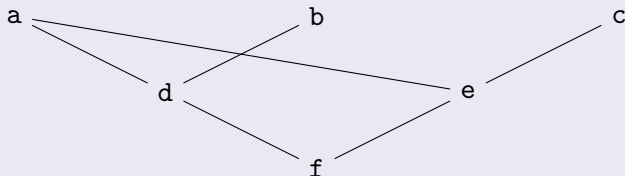
Loops Idêntica mas sem os primeiros duplicados.

CLOS Ordenação topológica do grafo de classes tendo em conta a ordenação local de superclasses.

Exemplo de Hierarquia de Classes

```
(defclass a () ())
(defclass b () ())
(defclass c () ())
(defclass d (a b) ())
(defclass e (a c) ())
(defclass f (d e) ())
```

Classes

Grafo de Herança das Classes

Lista de Precedências da Classe f

Flavors (f d a b e c standard-object t)

Loops (f d b e a c standard-object t)

CLOS (f d e a c b standard-object t)

Classes

MetaClasses

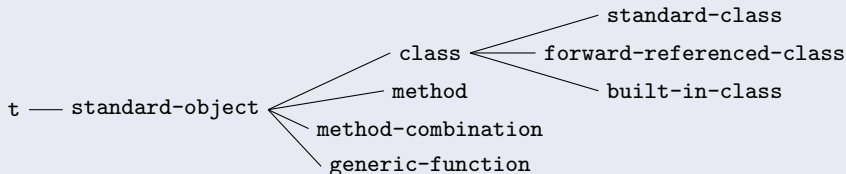
- As classes são representadas por objectos que são instâncias de classes.
- A **metaclasse** de um objecto é a classe da classe desse objecto.
- Uma **metaclasse** é uma classe cujas instâncias são classes.

Responsabilidades

- A metaclasse determina a forma de herança das classes que são suas instâncias.
- A metaclasse determina a representação das instâncias das classes que são suas instâncias.
- A metaclasse determina o acesso aos *slots* das instâncias.

Classes

Hierarquia de Classes

```
                                        class ──┬── standard-class
                                                ├── forward-referenced-class
                                                └── built-in-class
t ── standard-object ──┬── class
                       ├── method
                       ├── method-combination
                       └── generic-function
```

Definição

- A classe `t` não tem superclasse e é superclasse de todas as classes menos dela própria.
- A classe `standard-object` é subclasse directa da classe `t`, é uma instância da classe `standard-class` e é superclasse de todas as classes que são instâncias de `standard-class` menos dela própria.

Classes

A Metaclasse standard-class

```
> (defclass foo () ())          ;A 'normal' class
```

Classes

A Metaclasse standard-class

```
> (defclass foo () ())                ;A 'normal' class
#<STANDARD-CLASS FOO>
```

Classes

A Metaclasse standard-class

```
> (defclass foo () ())                ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                ;A 'normal' instance
```

Classes

A Metaclasse standard-class

```
> (defclass foo () ())              ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)              ;A 'normal' instance
#<FOO @ #x717910a2>                 ;Note #<CLASS INSTANCE>
```

Classes

A Metaclasse standard-class

```
> (defclass foo () ())                ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                ;A 'normal' instance
#<FOO @ #x717910a2>                   ;Note #<CLASS INSTANCE>
> (class-of (make-instance 'foo))
```

Classes

A Metaclasse standard-class

```
> (defclass foo () ())                ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                ;A 'normal' instance
#<FOO @ #x717910a2>                   ;Note #<CLASS INSTANCE>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                 ;Note #<METACLASS CLASS>
```

Classes

A Metaclasse standard-class

```
> (defclass foo () ())                ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                ;A 'normal' instance
#<FOO @ #x717910a2>                   ;Note #<CLASS INSTANCE>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                 ;Note #<METACLASS CLASS>
> (class-of (class-of (make-instance 'foo)))
```

Classes

A Metaclasse standard-class

```
> (defclass foo () ())                 ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                 ;A 'normal' instance
#<FOO @ #x717910a2>                    ;Note #<CLASS INSTANCE>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                  ;Note #<METACLASS CLASS>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>       ;STANDARD-CLASS metaclass
```

Classes

A Metaclasse standard-class

```
> (defclass foo () ())                 ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                 ;A 'normal' instance
#<FOO @ #x717910a2>                    ;Note #<CLASS INSTANCE>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                  ;Note #<METACLASS CLASS>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>   ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
```

Classes

A Metaclasse standard-class

```
> (defclass foo () ())                 ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                 ;A 'normal' instance
#<FOO @ #x717910a2>                    ;Note #<CLASS INSTANCE>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                  ;Note #<METACLASS CLASS>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>   ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
#<STANDARD-CLASS STANDARD-CLASS> ;Looping
```

A Metaclasse built-in-class

Classes

A Metaclasse standard-class

```
> (defclass foo () ())                   ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                   ;A 'normal' instance
#<FOO @ #x717910a2>                      ;Note #<CLASS INSTANCE>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                    ;Note #<METACLASS CLASS>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>   ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
#<STANDARD-CLASS STANDARD-CLASS> ;Looping
```

A Metaclasse built-in-class

```
> (class-of 1)
```

Classes

A Metaclasse standard-class

```
> (defclass foo () ())                  ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                  ;A 'normal' instance
#<FOO @ #x717910a2>                     ;Note #<CLASS INSTANCE>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                   ;Note #<METACLASS CLASS>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>   ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
#<STANDARD-CLASS STANDARD-CLASS> ;Looping
```

A Metaclasse built-in-class

```
> (class-of 1)
#<BUILT-IN-CLASS FIXNUM>                  ;Note #<METACLASS CLASS>
```

Classes

A Metaclasse standard-class

```
> (defclass foo () ())                ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                ;A 'normal' instance
#<FOO @ #x717910a2>                   ;Note #<CLASS INSTANCE>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                 ;Note #<METACLASS CLASS>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>      ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
#<STANDARD-CLASS STANDARD-CLASS> ;Looping
```

A Metaclasse built-in-class

```
> (class-of 1)
#<BUILT-IN-CLASS FIXNUM>                ;Note #<METACLASS CLASS>
> (class-of (class-of 1))               ;The metaclass of 1
```

Classes

A Metaclasse standard-class

```
> (defclass foo () ())                ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                ;A 'normal' instance
#<FOO @ #x717910a2>                   ;Note #<CLASS INSTANCE>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                 ;Note #<METACLASS CLASS>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>      ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
#<STANDARD-CLASS STANDARD-CLASS> ;Looping
```

A Metaclasse built-in-class

```
> (class-of 1)
#<BUILT-IN-CLASS FIXNUM>                ;Note #<METACLASS CLASS>
> (class-of (class-of 1))               ;The metaclass of 1
#<STANDARD-CLASS BUILT-IN-CLASS>        ;is BUILT-IN-CLASS
```

Classes

A Metaclasse standard-class

```
> (defclass foo () ())                  ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                  ;A 'normal' instance
#<FOO @ #x717910a2>                     ;Note #<CLASS INSTANCE>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                   ;Note #<METACLASS CLASS>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>        ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
#<STANDARD-CLASS STANDARD-CLASS> ;Looping
```

A Metaclasse built-in-class

```
> (class-of 1)
#<BUILT-IN-CLASS FIXNUM>                  ;Note #<METACLASS CLASS>
> (class-of (class-of 1))                 ;The metaclass of 1
#<STANDARD-CLASS BUILT-IN-CLASS>          ;is BUILT-IN-CLASS
> (class-of (class-of (class-of 1)));The metaclass of FIXNUM
```

Classes

A Metaclasse standard-class

```
> (defclass foo () ())                ;A 'normal' class
#<STANDARD-CLASS FOO>
> (make-instance 'foo)                ;A 'normal' instance
#<FOO @ #x717910a2>                   ;Note #<CLASS INSTANCE>
> (class-of (make-instance 'foo))
#<STANDARD-CLASS FOO>                 ;Note #<METACLASS CLASS>
> (class-of (class-of (make-instance 'foo)))
#<STANDARD-CLASS STANDARD-CLASS>      ;STANDARD-CLASS metaclass
> (class-of (class-of (class-of (make-instance 'foo))))
#<STANDARD-CLASS STANDARD-CLASS> ;Looping
```

A Metaclasse built-in-class

```
> (class-of 1)
#<BUILT-IN-CLASS FIXNUM>                ;Note #<METACLASS CLASS>
> (class-of (class-of 1))               ;The metaclass of 1
#<STANDARD-CLASS BUILT-IN-CLASS>        ;is BUILT-IN-CLASS
> (class-of (class-of (class-of 1)));The metaclass of FIXNUM
#<STANDARD-CLASS STANDARD-CLASS>        ;is STANDARD-CLASS
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())          ;The class baz is not defined yet...
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())        ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())        ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq baz-class
    (first (class-direct-superclasses (find-class 'bar))))
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())        ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq baz-class
    (first (class-direct-superclasses (find-class 'bar))))
#<FORWARD-REFERENCED-CLASS BAZ>  ;...but it exists already...
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())          ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq baz-class
    (first (class-direct-superclasses (find-class 'bar))))
#<FORWARD-REFERENCED-CLASS BAZ>  ;...but it exists already...
> (class-of baz-class)
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())        ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq baz-class
    (first (class-direct-superclasses (find-class 'bar))))
#<FORWARD-REFERENCED-CLASS BAZ>  ;...but it exists already...
> (class-of baz-class)
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())          ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq baz-class
    (first (class-direct-superclasses (find-class 'bar))))
#<FORWARD-REFERENCED-CLASS BAZ>  ;...but it exists already...
> (class-of baz-class)
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())           ;We now define baz...
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())          ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq baz-class
    (first (class-direct-superclasses (find-class 'bar))))
#<FORWARD-REFERENCED-CLASS BAZ>  ;...but it exists already...
> (class-of baz-class)
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())           ;We now define baz...
#<STANDARD-CLASS BAZ>
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())        ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq baz-class
    (first (class-direct-superclasses (find-class 'bar))))
#<FORWARD-REFERENCED-CLASS BAZ>  ;...but it exists already...
> (class-of baz-class)
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())          ;We now define baz...
#<STANDARD-CLASS BAZ>
> baz-class                     ;...and the saved class
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())          ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq baz-class
    (first (class-direct-superclasses (find-class 'bar))))
#<FORWARD-REFERENCED-CLASS BAZ>  ;...but it exists already...
> (class-of baz-class)
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())           ;We now define baz...
#<STANDARD-CLASS BAZ>
> baz-class                     ;...and the saved class
#<STANDARD-CLASS BAZ>           ;changes to a become a different thing
```

A função change-class

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())          ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq baz-class
    (first (class-direct-superclasses (find-class 'bar))))
#<FORWARD-REFERENCED-CLASS BAZ>  ;...but it exists already...
> (class-of baz-class)
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())           ;We now define baz...
#<STANDARD-CLASS BAZ>
> baz-class                     ;...and the saved class
#<STANDARD-CLASS BAZ>           ;changes to a become a different thing
```

A função change-class

```
> (setq foo-instance (make-instance 'foo)) ;;A normal instance
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())        ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq baz-class
    (first (class-direct-superclasses (find-class 'bar))))
#<FORWARD-REFERENCED-CLASS BAZ>  ;...but it exists already...
> (class-of baz-class)
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())          ;We now define baz...
#<STANDARD-CLASS BAZ>
> baz-class                    ;...and the saved class
#<STANDARD-CLASS BAZ>          ;changes to a become a different thing
```

A função change-class

```
> (setq foo-instance (make-instance 'foo)) ;;A normal instance
#<FOO @ #x717a0562>
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())         ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq baz-class
    (first (class-direct-superclasses (find-class 'bar))))
#<FORWARD-REFERENCED-CLASS BAZ>  ;...but it exists already...
> (class-of baz-class)
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())           ;We now define baz...
#<STANDARD-CLASS BAZ>
> baz-class                     ;...and the saved class
#<STANDARD-CLASS BAZ>           ;changes to a become a different thing
```

A função change-class

```
> (setq foo-instance (make-instance 'foo)) ;;A normal instance
#<FOO @ #x717a0562>
> (change-class foo-instance 'baz)      ;;Can we change its class?
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())          ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq baz-class
    (first (class-direct-superclasses (find-class 'bar))))
#<FORWARD-REFERENCED-CLASS BAZ>   ;...but it exists already...
> (class-of baz-class)
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())              ;We now define baz...
#<STANDARD-CLASS BAZ>
> baz-class                         ;...and the saved class
#<STANDARD-CLASS BAZ>               ;changes to a become a different thing
```

A função change-class

```
> (setq foo-instance (make-instance 'foo)) ;;A normal instance
#<FOO @ #x717a0562>
> (change-class foo-instance 'baz)        ;;Can we change its class?
#<BAZ @ #x717a0562>
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())          ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq baz-class
    (first (class-direct-superclasses (find-class 'bar))))
#<FORWARD-REFERENCED-CLASS BAZ>  ;...but it exists already...
> (class-of baz-class)
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())            ;We now define baz...
#<STANDARD-CLASS BAZ>
> baz-class                      ;...and the saved class
#<STANDARD-CLASS BAZ>            ;changes to a become a different thing
```

A função change-class

```
> (setq foo-instance (make-instance 'foo)) ;;A normal instance
#<FOO @ #x717a0562>
> (change-class foo-instance 'baz)      ;;Can we change its class?
#<BAZ @ #x717a0562>
> foo-instance
```

Classes

A Metaclasse forward-referenced-class

```
> (defclass bar (baz) ())         ;The class baz is not defined yet...
#<STANDARD-CLASS BAR>
> (setq baz-class
    (first (class-direct-superclasses (find-class 'bar))))
#<FORWARD-REFERENCED-CLASS BAZ>  ;...but it exists already...
> (class-of baz-class)
#<STANDARD-CLASS FORWARD-REFERENCED-CLASS>

> (defclass baz () ())          ;We now define baz...
#<STANDARD-CLASS BAZ>
> baz-class                     ;...and the saved class
#<STANDARD-CLASS BAZ>           ;changes to a become a different thing
```

A função change-class

```
> (setq foo-instance (make-instance 'foo)) ;;A normal instance
#<FOO @ #x717a0562>
> (change-class foo-instance 'baz)      ;;Can we change its class?
#<BAZ @ #x717a0562>
> foo-instance
#<BAZ @ #x717a0562>                     ;;Yes, we can!
```

Classes

Para se obter uma classe

- A partir de um objecto *foo*:
  (class-of *foo*)
- A partir do nome de um tipo '*bar*:
  (find-class '*bar*)

Exemplo

```
> (class-of "I am a string")
#<BUILT-IN-CLASS STRING>
> (find-class 'string)
#<BUILT-IN-CLASS STRING>
> (defclass foo () ())
#<STANDARD-CLASS FOO>
> (find-class 'foo)
#<STANDARD-CLASS FOO>
```

Classes

Nomes de Classes *vs* Classes

- Uma classe tem um nome (para melhor visualização).
- Um nome está associado a uma classe (para mais fácil acesso).
- Em geral:
  - (class-name (find-class *foo*))=*foo*
  - (find-class (class-name *foo*))=*foo*
- Mas pode-se mudar.

Exemplo

```
> (defclass foo () ())
#<STANDARD-CLASS FOO>
> (find-class 'foo)
#<STANDARD-CLASS FOO>
> (class-name (find-class 'foo))
FOO
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
> (setf (class-name (find-class 'foo)) 'bar) ;Change class name
BAR
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
> (setf (class-name (find-class 'foo)) 'bar) ;Change class name
BAR
> my-foo                                    ;Our instance is the same
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
> (setf (class-name (find-class 'foo)) 'bar) ;Change class name
BAR
> my-foo                            ;Our instance is the same
#<BAR @ #x71788672>
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
> (setf (class-name (find-class 'foo)) 'bar) ;Change class name
BAR
> my-foo                              ;Our instance is the same
#<BAR @ #x71788672>
> (make-instance 'foo)                ;foo references the same class
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
> (setf (class-name (find-class 'foo)) 'bar) ;Change class name
BAR
> my-foo                              ;Our instance is the same
#<BAR @ #x71788672>
> (make-instance 'foo)                ;foo references the same class
#<BAR @ #x715ea4b2>
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
> (setf (class-name (find-class 'foo)) 'bar) ;Change class name
BAR
> my-foo                              ;Our instance is the same
#<BAR @ #x71788672>
> (make-instance 'foo)                ;foo references the same class
#<BAR @ #x715ea4b2>
> (make-instance 'bar)
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
> (setf (class-name (find-class 'foo)) 'bar) ;Change class name
BAR
> my-foo                               ;Our instance is the same
#<BAR @ #x71788672>
> (make-instance 'foo)                 ;foo references the same class
#<BAR @ #x715ea4b2>
> (make-instance 'bar)
Error No class named: BAR.             ;but bar doesn't
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
> (setf (class-name (find-class 'foo)) 'bar) ;Change class name
BAR
> my-foo                              ;Our instance is the same
#<BAR @ #x71788672>
> (make-instance 'foo)                ;foo references the same class
#<BAR @ #x715ea4b2>
> (make-instance 'bar)
Error No class named: BAR.            ;but bar doesn't
> (setf (find-class 'bar) (find-class 'foo)) ;Now it does
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
> (setf (class-name (find-class 'foo)) 'bar) ;Change class name
BAR
> my-foo                               ;Our instance is the same
#<BAR @ #x71788672>
> (make-instance 'foo)                 ;foo references the same class
#<BAR @ #x715ea4b2>
> (make-instance 'bar)
Error No class named: BAR.             ;but bar doesn't
> (setf (find-class 'bar) (find-class 'foo)) ;Now it does
#<STANDARD-CLASS BAR>
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
> (setf (class-name (find-class 'foo)) 'bar) ;Change class name
BAR
> my-foo                             ;Our instance is the same
#<BAR @ #x71788672>
> (make-instance 'foo)               ;foo references the same class
#<BAR @ #x715ea4b2>
> (make-instance 'bar)
Error No class named: BAR.           ;but bar doesn't
> (setf (find-class 'bar) (find-class 'foo)) ;Now it does
#<STANDARD-CLASS BAR>
> (make-instance 'bar)              ;bar is the same class...
#<BAR @ #x717c874a>
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
> (setf (class-name (find-class 'foo)) 'bar) ;Change class name
BAR
> my-foo                               ;Our instance is the same
#<BAR @ #x71788672>
> (make-instance 'foo)                 ;foo references the same class
#<BAR @ #x715ea4b2>
> (make-instance 'bar)
Error No class named: BAR.             ;but bar doesn't
> (setf (find-class 'bar) (find-class 'foo)) ;Now it does
#<STANDARD-CLASS BAR>
> (make-instance 'bar)                ;bar is the same class...
#<BAR @ #x717c874a>
> (make-instance 'foo)                ;...as foo
#<BAR @ #x717cef6a>
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
> (setf (class-name (find-class 'foo)) 'bar) ;Change class name
BAR
> my-foo                               ;Our instance is the same
#<BAR @ #x71788672>
> (make-instance 'foo)                 ;foo references the same class
#<BAR @ #x715ea4b2>
> (make-instance 'bar)
Error No class named: BAR.             ;but bar doesn't
> (setf (find-class 'bar) (find-class 'foo)) ;Now it does
#<STANDARD-CLASS BAR>
> (make-instance 'bar)                ;bar is the same class...
#<BAR @ #x717c874a>
> (make-instance 'foo)                ;...as foo
#<BAR @ #x717cef6a>
> (setf (find-class 'foo) nil)        ;foo doesn't reference the class
NIL
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
> (setf (class-name (find-class 'foo)) 'bar) ;Change class name
BAR
> my-foo                               ;Our instance is the same
#<BAR @ #x71788672>
> (make-instance 'foo)                 ;foo references the same class
#<BAR @ #x715ea4b2>
> (make-instance 'bar)
Error No class named: BAR.             ;but bar doesn't
> (setf (find-class 'bar) (find-class 'foo)) ;Now it does
#<STANDARD-CLASS BAR>
> (make-instance 'bar)                ;bar is the same class...
#<BAR @ #x717c874a>
> (make-instance 'foo)                ;...as foo
#<BAR @ #x717cef6a>
> (setf (find-class 'foo) nil)        ;foo doesn't reference the class
NIL
> (make-instance 'foo)                ;so it can't be used
```

Classes

Nomes de Classes *vs* Classes

```
> (setf my-foo (make-instance 'foo));Let's save a foo instance
#<FOO @ #x71788672>
> (setf (class-name (find-class 'foo)) 'bar) ;Change class name
BAR
> my-foo                                    ;Our instance is the same
#<BAR @ #x71788672>
> (make-instance 'foo)                      ;foo references the same class
#<BAR @ #x715ea4b2>
> (make-instance 'bar)
Error No class named: BAR.                  ;but bar doesn't
> (setf (find-class 'bar) (find-class 'foo)) ;Now it does
#<STANDARD-CLASS BAR>
> (make-instance 'bar)                     ;bar is the same class...
#<BAR @ #x717c874a>
> (make-instance 'foo)                     ;...as foo
#<BAR @ #x717cef6a>
> (setf (find-class 'foo) nil)             ;foo doesn't reference the class
NIL
> (make-instance 'foo)                     ;so it can't be used
Error No class named: FOO.
```

Classes

A Função Genérica make-instance

```
(defgeneric make-instance (class &rest initargs))
```

O Método especializado para **símbolos**

```
(defmethod make-instance ((class symbol) &rest initargs)
  (apply #'make-instance (find-class class) initargs))
```

O Método especializado para **classes**

```
(defmethod make-instance ((class class) &rest initargs)
  (let ((instance (apply #'allocate-instance class initargs)))
    (apply #'initialize-instance instance initargs)
    instance))
```

O Optimizador

```
(define-compiler-macro make-instance (class-expr &rest init-exprs)
  (if (and (consp class-expr) (eq (first class-expr) 'quote))
    (make-instance->constructor-call (second class-expr) init-exprs)
    ...))
```

Classes

Exemplo: Classes Anónimas

- Criar uma classe com um nome único.
- Criar uma instância a partir dessa classe.

A Macro anonymous-class

```
(defmacro anonymous-class (supers slots &rest options)
  `(defclass ,(gensym) ,supers ,slots ,@options))
```

Exemplo

```
> (draw (make-instance (anonymous-class (color-mixin circle)
                          ((filled? :initform t :reader filled?)))
           :color :blue
           :radius 3)
        (make-instance 'colored-printer))
changing printer ink color to BLUE
drawing an circle on printer!
changing printer ink color to BLACK
```

Classes

Slots

1. A expressão `(slot-value` *obj nome*`)` devolve o valor do slot *nome* no *obj*.
2. Se não existir o slot, invoca a função genérica `slot-missing`:
   `(slot-missing (class-of` *obj*`)` *obj nome* `'slot-value)`
3. Se existir o slot mas estiver sem valor, invoca a função genérica `slot-unbound`:
   `(slot-unbound (class-of` *obj*`)` *obj nome*`)`
4. A expressão `(setf (slot-value` *obj nome*`)` *novo-valor*`)` altera o valor do slot *nome* no *obj*.
5. Se não existir o slot, invoca a função genérica `slot-missing`:
   `(slot-missing (class-of` *obj*`)` *obj nome* `'setf` *novo-valor*`)`

Classes

Slots

```
(defclass foo ()
  ((slot1)))

> (setq my-foo (make-instance 'foo))
#<FOO @ #x71648d6a>
> (* (slot-value my-foo 'slot1) 2)
```

Classes

Slots

```
(defclass foo ()
  ((slot1)))

> (setq my-foo (make-instance 'foo))
#<FOO @ #x71648d6a>
> (* (slot-value my-foo 'slot1) 2)
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]
```

Classes

Slots

```
(defclass foo ()
  ((slot1)))

> (setq my-foo (make-instance 'foo))
#<FOO @ #x71648d6a>
> (* (slot-value my-foo 'slot1) 2)
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [ABORT] Return to SLIME's top level.
```

Classes

Slots

```
(defclass foo ()
  ((slot1)))

> (setq my-foo (make-instance 'foo))
#<FOO @ #x71648d6a>
> (* (slot-value my-foo 'slot1) 2)
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [ABORT] Return to SLIME's top level.

:C 2
```

Classes

Slots

```
(defclass foo ()
  ((slot1)))

> (setq my-foo (make-instance 'foo))
#<FOO @ #x71648d6a>
> (* (slot-value my-foo 'slot1) 2)
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [ABORT] Return to SLIME's top level.

:C 2
enter expression which will evaluate to a value to use: 25
```

Classes

Slots

```
(defclass foo ()
  ((slot1)))

> (setq my-foo (make-instance 'foo))
#<FOO @ #x71648d6a>
> (* (slot-value my-foo 'slot1) 2)
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [ABORT] Return to SLIME's top level.

:C 2
enter expression which will evaluate to a value to use: 25
```

Classes

Slots

```
(defclass foo ()
  ((slot1)))

> (setq my-foo (make-instance 'foo))
#<FOO @ #x71648d6a>
> (* (slot-value my-foo 'slot1) 2)
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [ABORT] Return to SLIME's top level.

:C 2
enter expression which will evaluate to a value to use: 25
50
```

Classes

Slots

```
(defclass foo ()
  ((slot1)))

> (setq my-foo (make-instance 'foo))
#<FOO @ #x71648d6a>
> (* (slot-value my-foo 'slot1) 2)
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [ABORT] Return to SLIME's top level.

:C 2
enter expression which will evaluate to a value to use: 25
50
CL-USER> (slot-value my-foo 'slot1)
```

Classes

Slots

```
(defclass foo ()
  ((slot1)))

> (setq my-foo (make-instance 'foo))
#<FOO @ #x71648d6a>
> (* (slot-value my-foo 'slot1) 2)
The slot SLOT1 is unbound in the object #<FOO @ #x7161dfaa> of class
#<STANDARD-CLASS FOO>.
   [Condition of type UNBOUND-SLOT]

Restarts:
 0: [TRY-AGAIN] Try accessing the slot again
 1: [USE-VALUE] Return a value
 2: [STORE-VALUE] Store a value and return it
 3: [ABORT] Return to SLIME's top level.

:C 2
enter expression which will evaluate to a value to use: 25
50
CL-USER> (slot-value my-foo 'slot1)
25
```

Classes

Slots

- `slot-value` e `(setf slot-value)` são **funções**...
- ... mas **não são funções genéricas**.
- Mas, nas implementações que incluem o MOP (todas, actualmente), invocam as funções genéricas `slot-value-using-class` e `(setf slot-value-using-class)`

A Função (não-genérica) `slot-value`

```
(defun slot-value (object slot-name)
  (let* ((class (class-of object))
         (slot-definition (find-slot-definition class slot-name)))
    (if (null slot-definition)
      (slot-missing class object slot-name 'slot-value)
      (slot-value-using-class class object slot-definition))))
```

Classes

A Função Genérica slot-value-using-class

```
(defmethod slot-value-using-class
    ((class standard-class)
     (object standard-object)
     (slotd standard-effective-slot-definition))
  (if ...
    (slot-unbound class object (slot-definition-name slotd))
    ...))
```

A Função Genérica slot-unbound

```
(defmethod slot-unbound ((class t) instance slot-name)
  (restart-case
      (error 'unbound-slot :name slot-name :instance instance)
    (use-value (value)
      ...)
    (store-value (new-value)
      ...)))
```

Protocolo de Objecto

Protocolo

Modelo abstracto do *comportamento* de um sistema.

Protocolo

Conjunto de funções genéricas que colaboram para um mesmo fim.

Protocolos em CLOS

- Criação e inicialização de instância
- Reinicialização de instância
- Mudança da classe de instância
- Redefinição de classes
- Acesso a slot de instância
- Invocação de função genérica

Protocolo de Objecto

Criação de Instância

1. Combinar inicializações explícitas (`make-instance`) com valores de omissão (`:default-initargs` e `:initforms`).
2. Verificar a validade das inicializações.
3. Alocação de espaço físico para a instância (`allocate-instance`).
4. Preenchimento dos *slots* usando as inicializações (`initialize-instance` e `shared-initialize`).

Protocolo de Objecto

Criação de Instância

- `make-instance` invoca `allocate-instance` e, depois, `initialize-instance`.
- `allocate-instance` aloca o espaço físico para a instância.
- `initialize-instance` invoca `shared-initialize`.
- `shared-initialize` atribui os *slots* com base nos `:initargs`, `:default-initargs` e `:initforms`.

Protocolo de Objecto

Criação de Instância - make-instance

```
(defmethod make-instance ((class class) &rest initargs)

  ;; Verify initialization validity

  (let ((instance (apply #'allocate-instance class initargs)))
    (apply #'initialize-instance instance initargs)
    instance))
```

Criação de Instância - initialize-instance

```
(defmethod initialize-instance ((instance standard-object)
                                &rest initargs &key)
  (apply #'shared-initialize instance t initargs))
```

Protocolo de Objecto

Mudança da Classe de Instância

1. Modificação da estrutura da instância por adição de novos slots e eliminação dos não existentes na futura classe.
2. Preenchimento dos *slots* novos usando as inicializações (`update-instance-for-different-class` e `shared-initialize`).

Mudança da Classe de Instância

- `change-class` modifica um objecto para ser uma instância de uma classe diferente.
- `change-class` invoca `update-instance-for-different-class`.
- `update-instance-for-different-class` invoca `shared-initialize`.

Protocolo de Objecto

Mudança da Classe de Instância

```
(defmethod change-class ((instance standard-object)
                         (new-class standard-class)
                         &rest initargs &key)
  (let* ((old-class (class-of instance))
         (new-instance (allocate-instance new-class))
         (old-slots (get-slots instance))
         (new-slots (get-slots new-instance)))
    ;; Copy shared slots
    ;; Make the old instance point to the new storage.
    (apply #'update-instance-for-different-class
           new-instance
           instance
           initargs)
    instance))

(defmethod update-instance-for-different-class
    ((previous standard-object) (current standard-object)
     &rest initargs &key)
    ...
    (apply #'shared-initialize current added-slots initargs)))
```

Protocolo de Objecto

Redefinição da Classe de Instância

1. Modificação da estrutura da classe já existente.
2. Se ocorrer adição e/ou remoção de *slots* e/ou alteração da ordem dos *slots*, as instâncias já existentes são actualizadas (num instante indeterminado mas antes de qualquer acesso aos *slots*).
3. Para cada instância, modificação da estrutura da instância por adição de novos slots e eliminação dos não existentes na futura classe.
4. Preenchimento dos *slots* novos usando as inicializações (`update-instance-for-redefined-class` e `shared-initialize`).

Protocolo de Objecto

Redefinição da Classe de Instância

- `make-instances-obsolete` modifica os objectos para reflectirem a nova definição da classe.
- `make-instances-obsolete` invoca (num instance indeterminado) `update-instance-for-redefined-class` para cada instância.
- `update-instance-for-redefined-class` invoca `shared-initialize`.
- `shared-initialize` atribui os *slots* com base nos `:initargs`, `:default-initargs` e `:initforms`.

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defclass complex-number ()   ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))
```

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defclass complex-number ()   ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
```

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defclass complex-number ()    ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
> (setq 3+4i (make-instance 'complex-number :real 3 :imag 4))
#<COMPLEX-NUMBER @ #x717816b2>
```

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defclass complex-number ()    ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
> (setq 3+4i (make-instance 'complex-number :real 3 :imag 4))
#<COMPLEX-NUMBER @ #x717816b2>
> (slot-value 1+2i 'real)
1
```

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defclass complex-number ()   ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
> (setq 3+4i (make-instance 'complex-number :real 3 :imag 4))
#<COMPLEX-NUMBER @ #x717816b2>
> (slot-value 1+2i 'real)
1

(defclass complex-number ()   ;Redefine (polar) complex-number
  ((rho :initarg :rho)
   (theta :initarg :theta)))
```

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defclass complex-number ()    ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
> (setq 3+4i (make-instance 'complex-number :real 3 :imag 4))
#<COMPLEX-NUMBER @ #x717816b2>
> (slot-value 1+2i 'real)
1

(defclass complex-number ()    ;Redefine (polar) complex-number
  ((rho :initarg :rho)
   (theta :initarg :theta)))

> (slot-value 1+2i 'real)      ;The slot 'real' is gone
```

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defclass complex-number ()    ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
> (setq 3+4i (make-instance 'complex-number :real 3 :imag 4))
#<COMPLEX-NUMBER @ #x717816b2>
> (slot-value 1+2i 'real)
1

(defclass complex-number ()    ;Redefine (polar) complex-number
  ((rho :initarg :rho)
   (theta :initarg :theta)))

> (slot-value 1+2i 'real)     ;The slot 'real' is gone
The slot REAL is missing in the object #<COMPLEX-NUMBER @ #x717705a2>
```

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defclass complex-number ()   ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
> (setq 3+4i (make-instance 'complex-number :real 3 :imag 4))
#<COMPLEX-NUMBER @ #x717816b2>
> (slot-value 1+2i 'real)
1

(defclass complex-number ()   ;Redefine (polar) complex-number
  ((rho :initarg :rho)
   (theta :initarg :theta)))

> (slot-value 1+2i 'real)     ;The slot 'real' is gone
The slot REAL is missing in the object #<COMPLEX-NUMBER @ #x717705a2>

> (slot-value 1+2i 'rho)      ;The slot 'rho' is unbound
```

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defclass complex-number ()    ;Define (rectangular) complex-number
  ((real :initarg :real)
   (imag :initarg :imag)))

> (setq 1+2i (make-instance 'complex-number :real 1 :imag 2))
#<COMPLEX-NUMBER @ #x717705a2>
> (setq 3+4i (make-instance 'complex-number :real 3 :imag 4))
#<COMPLEX-NUMBER @ #x717816b2>
> (slot-value 1+2i 'real)
1

(defclass complex-number ()    ;Redefine (polar) complex-number
  ((rho :initarg :rho)
   (theta :initarg :theta)))

> (slot-value 1+2i 'real)      ;The slot 'real' is gone
The slot REAL is missing in the object #<COMPLEX-NUMBER @ #x717705a2>

> (slot-value 1+2i 'rho)       ;The slot 'rho' is unbound
The slot RHO is unbound in the object #<COMPLEX-NUMBER @ #x717705a2>
```

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defmethod update-instance-for-redefined-class :before
    ((c complex-number)
     added-slots
     discarded-slots
     property-list                  ;(real 3 imag 4)
     &rest args
     &key &allow-other-keys)
```

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defmethod update-instance-for-redefined-class :before
    ((c complex-number)
     added-slots
     discarded-slots
     property-list                  ;(real 3 imag 4)
     &rest args
     &key &allow-other-keys)
  (let ((r (getf property-list 'real))
        (i (getf property-list 'imag)))
    (setf (slot-value c 'rho)
            (sqrt (+ (* r r) (* i i)))
          (slot-value c 'theta)
            (atan i r))))
```

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defmethod update-instance-for-redefined-class :before
    ((c complex-number)
     added-slots
     discarded-slots
     property-list              ;(real 3 imag 4)
     &rest args
     &key &allow-other-keys)
  (let ((r (getf property-list 'real))
        (i (getf property-list 'imag)))
    (setf (slot-value c 'rho)
            (sqrt (+ (* r r) (* i i)))
          (slot-value c 'theta)
            (atan i r))))

> (slot-value 1+2i 'rho) ;Too late for the first instance
```

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defmethod update-instance-for-redefined-class :before
    ((c complex-number)
     added-slots
     discarded-slots
     property-list                ;(real 3 imag 4)
     &rest args
     &key &allow-other-keys)
  (let ((r (getf property-list 'real))
        (i (getf property-list 'imag)))
    (setf (slot-value c 'rho)
            (sqrt (+ (* r r) (* i i)))
          (slot-value c 'theta)
            (atan i r))))

> (slot-value 1+2i 'rho) ;Too late for the first instance
The slot RHO is unbound in the object #<COMPLEX-NUMBER @ #x717705a2>
```

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defmethod update-instance-for-redefined-class :before
    ((c complex-number)
     added-slots
     discarded-slots
     property-list              ;(real 3 imag 4)
     &rest args
     &key &allow-other-keys)
  (let ((r (getf property-list 'real))
        (i (getf property-list 'imag)))
    (setf (slot-value c 'rho)
            (sqrt (+ (* r r) (* i i)))
          (slot-value c 'theta)
            (atan i r))))

> (slot-value 1+2i 'rho) ;Too late for the first instance
The slot RHO is unbound in the object #<COMPLEX-NUMBER @ #x717705a2>

> (slot-value 3+4i 'rho) ;But on time for the second one
```

Protocolo de Objecto

Redefinição da Classe de Instância

```
(defmethod update-instance-for-redefined-class :before
    ((c complex-number)
     added-slots
     discarded-slots
     property-list                    ;(real 3 imag 4)
     &rest args
     &key &allow-other-keys)
  (let ((r (getf property-list 'real))
        (i (getf property-list 'imag)))
    (setf (slot-value c 'rho)
            (sqrt (+ (* r r) (* i i)))
          (slot-value c 'theta)
            (atan i r))))

> (slot-value 1+2i 'rho) ;Too late for the first instance
The slot RHO is unbound in the object #<COMPLEX-NUMBER @ #x717705a2>

> (slot-value 3+4i 'rho) ;But on time for the second one
5.0
```

Protocolo de Meta Objecto

Exemplo hipotético: Acesso a um *slot*

- Uma instância é representada por um *array*.
- O primeiro elemento do *array* é a classe a que a instância pertence.
- Os restantes elementos do *array* são os valores dos slots.

A função slot-value

```
(defun slot-value (instance slot-name)
  (let ((class (aref instance 0)))
    (let ((slots (class-slots class)))
      (aref instance
            (1+ (position slot-name slots))))))
```

Problema

Inflexível: uma única representação de instância

Protocolo de Meta Objecto

Exemplo hipotético: Acesso a um *slot*

- A solução consiste em delegar a *interpretação* do acesso a um *slot* noutra entidade.
- Uma possibilidade: usar a classe da classe da instância (i.e., a metaclasse da instância).
- A metaclasse *intermedeia* o acesso à instância.

A função slot-value

```
(defun slot-value (instance slot-name)
  (slot-value-using-class (class-of instance)
                          instance
                          slot-name))
```

Protocolo de Meta Objecto

Exemplo hipotético: Acesso a um *slot*

- Para a metaclass de omissão (por exemplo, *default-class*), uma instância é representada por um *array*.

A função slot-value

```
(defmethod slot-value-using-class ((class default-class)
                                    instance
                                    slot-name)
  (let ((slots (class-slots class)))
    (aref instance
          (1+ (position slot-name slots)))))
```

Protocolo de Meta Objecto

Exemplo hipotético: Acesso a um *slot*

- Para outra metaclasse (por exemplo, *hash-table-class*), uma instância é representada por uma *hash-table*.

A função slot-value

```
(defmethod slot-value-using-class ((class hash-table-class)
                                   instance
                                   slot-name)
  (gethash instance slot-name))
```

Protocolo de Meta Objecto

Exemplo real: Acesso a um *slot*

- Para a metaclasse de omissão *standard-class*.

A função slot-value

```
(defun slot-value (object slot-name)
  (let* ((class (class-of object))
         (slot-definition (find-slot-definition class slot-name)))
    (if (null slot-definition)
      (slot-missing class object slot-name 'slot-value)
      (slot-value-using-class class object slot-definition))))
```

A função slot-value-using-class

```
(defmethod slot-value-using-class
    ((class standard-class)
     (object standard-object)
     (slotd standard-effective-slot-definition))
  (if ...
    (slot-unbound class object (slot-definition-name slotd))
    ...))
```